CircuitSAT to SAT

Samuel Mathias Reng Henning 201504481

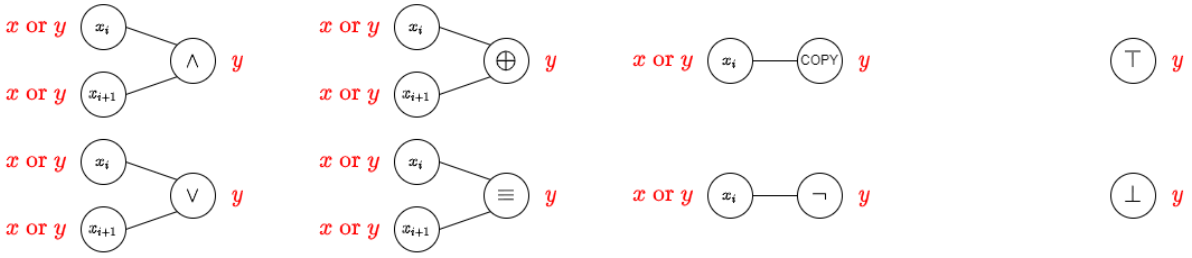
April 18, 2022

Exercise 1: Implement the reduction from CircuitSAT to SAT

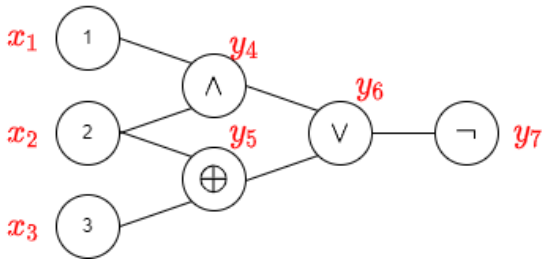
I will give a quick intro to circuits, to provide terminology for first showing the effectiveness of the reduction and then the maximum polynomial efficiency.

Circuits and the circuit file format

A circuit is a type of directed graph, with 8 types of nodes, called gates. As the word 'gates' signifies, this is logical gates. A gate has 0 to 2 inputs, computes a logical operation, and distribute the Boolean result through all outgoing arcs. The various gates are shown in the illustration below, with inputs signified as x and gate-outputs signified as y



Below I show an example of a graph and the corresponding text-file:




```
3
A 1 2
X 2 3
O 4 5
N 6
```

Parsing the circuit file

The circuit graph is provided as a text-file with the following structure:

- First line holds an integer denoting the number of input gates, and nothing else.
- Each subsequent line holds a node, that is a gate, consisting of:
  1. Type of node from 0,1,A,O,X,E,N,C specifying a FALSE, TRUE, AND, OR, XOR, EQUALITY, NOT or COPY-gate
  2. The inputs to the gate, specified by the each input-gates integer number.

  Note:

  – The inputs are numbered 1 through n. The other gates are numbered implicitly by their line number, starting from n+1.

  – The output of the circuit is the last gate.

In the end of this section I have placed a code-pic of the *read_circuit_file* function. In parsing the circuit file the `read_circuit_file` function checks:

- that the first line contains a sequence of characters that is not split by any white-space, (code line 127)
- that the first line contains a sequence of characters that corresponds to an integer,(code line 129)
- that there are at least one gate-line (code line 132)
- that each gate-line corresponds to a real gate-type with the correct number of inputs, where each input is a previous gate (including input-gates). (code line 135-149)

Thus I know the input is correctly formed and thus that the Circuit object is well-formed in later parts of the code.

```
121     try:
122         file = open(fname, 'r')
123         lines = file.readlines()
124         file.close()
125
126         first_line = lines[0].split()
127         if len(first_line) != 1:
128             raise ValueError
129         number_of_inputs = int(first_line[0])
130
131         gates = [[character for character in line.split()] for line in lines[1:]]
132         if len(gates) < 1:
133             raise ValueError
134
135         number_of_gates_already_described = number_of_inputs
136         for gate_line in gates:
137             if not valid_gate_type(gate_line[0]):
138                 raise ValueError
139             elif is_nullary(gate_line[0]) and len(gate_line) != 1:
140                 raise ValueError
141             elif is_unary(gate_line[0]) and (len(gate_line) != 2 or
142                                              1 > int(gate_line[1]) > number_of_gates_already_described):
143                 raise ValueError
144             elif is_binary(gate_line[0]) and (len(gate_line) != 3 or
145                                               1 > int(gate_line[1]) > number_of_gates_already_described or
146                                               1 > int(gate_line[2]) > number_of_gates_already_described):
147                 raise ValueError
148             else:
149                 number_of_gates_already_described += 1
150
151         return Circuit(number_of_inputs, gates)
152     except ValueError:
153         return 'INVALID'
154
```

0.1 Effectiveness of the reduction

First I will show the reduction approach and then I show that the implementation obeys this method.

Our goal is to transform a circuit, on the form described above, into a cnf-formula, while preserving satisfiability and non-satisfiability. The approach used will introduce an additional variable for each gate, and enforce that this gate-variable is equal to the output of the gate, depending on the gate-type and the gate inputs.

We have eight gate types, each with their gate-variable. I will list the gate truth tables along with the possible truth-values of the gate-variable. Afterwards I will show how to determine the cnf-clauses that enforce the truth tables.

x_1	x_2	y	$x_1 \wedge x_2$	$x_1 \wedge x_2 \equiv y$
0	0	0	0	1
0	**0**	**1**	**0**	**0**
0	1	0	0	1
0	**1**	**1**	**0**	**0**
1	0	0	0	1
1	**0**	**1**	**0**	**0**
1	**1**	**0**	**1**	**0**
1	1	1	1	1

x_1	x_2	y	$x_1 \vee x_2$	$x_1 \vee x_2 \equiv y$
0	0	0	0	1
0	**0**	**1**	**0**	**0**
0	**1**	**0**	**1**	**0**
0	1	1	1	1
1	**0**	**0**	**1**	**0**
1	0	1	1	1
1	**1**	**0**	**1**	**0**
1	1	1	1	1

x_1	x_2	y	$x_1 \oplus x_2$	$x_1 \oplus x_2 \equiv y$
0	0	0	0	1
0	**0**	**1**	**0**	**0**
0	**1**	**0**	**1**	**0**
0	1	1	1	1
1	**0**	**0**	**1**	**0**
1	0	1	1	1
1	1	0	0	1
1	**1**	**1**	**0**	**0**

x_1	x_2	y	$x_1 = x_2$	$x_1 = x_2 \equiv y$
0	**0**	**0**	**1**	**0**
0	0	1	1	1
0	1	0	0	1
0	**1**	**1**	**0**	**0**
1	0	0	0	1
1	**0**	**1**	**0**	**0**
1	**1**	**0**	**1**	**0**
1	1	1	1	1

x_1	y	$\neg x_1$	$\neg x_1 \equiv y$
0	**0**	**1**	**0**
0	1	1	1
1	0	0	1
1	**1**	**0**	**0**

x_1	y	$COPY x_1$	$COPY x_1 \equiv y$
0	0	0	1
0	**1**	**0**	**0**
1	**0**	**1**	**0**
1	1	1	1

$\top$	y	$\top \equiv y$
1	**0**	**0**
1	1	1

$\bot$	y	$\bot \equiv y$
0	0	1
0	**1**	**0**

When translating a truth table to cnf one will use all the rows where the result is false. For each of these rows, the corresponding clause is the same variables, with 'reversed truth values', such that the row:

x_1	x_2	y	$x_1 \wedge x_2$	$x_1 \wedge x_2 \equiv y$
0	**0**	**1**	**0**	**0**

becomes

$$x_1 \vee x_2 \vee \neg y$$

This is only one row, and so to enforce the whole truth table we get

$$(x_1 \vee x_2 \vee \neg y) \wedge$$
$$(x_1 \vee \neg x_2 \vee \neg y) \wedge$$
$$(\neg x_1 \vee x_2 \vee \neg y) \wedge$$
$$(\neg x_1 \vee \neg x_2 \vee y)$$

Now y must hold the value of the AND-gate for all these clause to be true. I have produced such a set of clauses for each gate-type, and written them into the code in the method `CSAT_gate_to_SAT_clause`, as seen below.

```
156 def CSAT_gate_to_SAT_clause(y, gate):
157     match gate[0]:
158         case '0':
159             return [[-y]]
160         case '1':
161             return [[y]]
162         case 'A':
163             return [[int(gate[1]), int(gate[2]), -y],
164                     [int(gate[1]), -int(gate[2]), -y],
165                     [-int(gate[1]), int(gate[2]), -y],
166                     [-int(gate[1]), -int(gate[2]), y]]
167         case 'O':
168             return [[int(gate[1]), int(gate[2]), -y],
169                     [int(gate[1]), -int(gate[2]), y],
170                     [-int(gate[1]), int(gate[2]), y],
171                     [-int(gate[1]), -int(gate[2]), y]]
172         case 'N':
173             return [[int(gate[1]), y],
174                     [-int(gate[1]), -y]]
175         case 'C':
176             return [[int(gate[1]), -y],
177                     [-int(gate[1]), y]]
178         case 'X':
179             return [[int(gate[1]), int(gate[2]), -y],
180                     [int(gate[1]), -int(gate[2]), y],
181                     [-int(gate[1]), int(gate[2]), y],
182                     [-int(gate[1]), -int(gate[2]), -y]]
183         case 'E':
184             return [[int(gate[1]), int(gate[2]), y],
185                     [int(gate[1]), -int(gate[2]), -y],
186                     [-int(gate[1]), -int(gate[2]), -y],
187                     [-int(gate[1]), -int(gate[2]), y]]
188
```

Now the reduction consist of taking one gate at a time and transforming it, thus enforcing that the gate-variable holds the value of the logic-gate (see line 194-197 in the code-pic below). However after doing so we must ensure that the last gate-variable must be true - otherwise the cnf will be true whenever the inputs and outputs obeys the truth-tables, even if the last output is false. Therefore we add a clause containing only the last gate-variable, thus ensuring that it must be true for the cnf to accept (line 198).

```
194     cnf = []
195     for i, gate in enumerate(C.gates):
196         y = C.n+1+i
197         cnf.extend(CSAT_gate_to_SAT_clause(y, gate))
198     cnf.append([y])
199     return cnf
```

Thus the SAT problem cnf, made by reducing a CircuitSAT problem, maintains satisfiability and non-satisfiability of the original problem.

I will process to show that this reduction is performed within polynomial time. When reading the problem into memory I first iterate over all lines once, to read each line into a string object, and then again to construct an array for each gate-line, and a third time for checking that the document holds the correct structure. This makes up for a time-complexity of $3 \times \#Gates$ meaning $O(\#Gates)$. To transform the circuit I iterate over each gate once again, and then I am done. Thus the time-complexity is best described in the number of gates, but can be transformed to a time-complexity over a maximum number of inputs. For n inputs the maximum number of gates are $\frac{n^2}{2}$. Thus the time-complexity is in the order of $O(n^2)$, thus polynomial.

Exercise 2: CircuitSAT$_{\geq 2}$ to SAT

Before I go into how I did it I wanna adress what I thought to be a point of ambiguity. Take the following four cnfs

$$true \qquad true \wedge true \qquad true \vee false \qquad true \oplus false$$

All of them are clearly true, and non of them can be set to false, as there are no variables. Should such cases be counted as a true CircuitSAT$_{\geq 2}$? The argument that this is *not* a CircuitSAT$_{\geq 2}$ is that there are no variables to set, thus there are only one assignment: 'nothing'. The argument that this *is* a CircuitSAT$_{\geq 2}$ is that no matter how you set the variables the circuit will be true. As the exercise doesn't specify I have picked, and I chose the later; if there are no variables and the circuit evaluates to true, then it is taken as true for an infinite amount of assignments, and vise versa, if false then false for an infinite amount of assignments.

Now to the approach I used. We already have a way to transform a CircuitSAT-circuit int a SAT-cnf; lets call this cnf the 'first cnf'. I make a 'second cnf' by taking a copy of the first cnf and adding the number of variables from the first-cnf to each variable number of the second, thus changing the variable numbers so as to start after the last variable-number of the first cnf. Now there are two copies of the instance, both of which must be true for the SAT-problem to be satisfiable. However both can still be sat to the same assignment! Thus adding nothing of significance yet.

Therefore I will add that all the corresponding input variables cannot be identical. I do this by taking the first variable of the first instance and the first variable of the second instance as inputs to an equal-gate; I do the same for the second variable of each, and then the third and so on. I then gather the results of the equal gates into one by adding all the equal-gates using AND gates. I also continue adding the outputs of the and-gates in an iterative manner until there is only one output from these. This one output will be true if all corresponding variables from the two instances are equal. I add a NOT-gate to this output, which will then be true if one or more corresponding outputs are non-equal. I lastly add the output of the NOT-gate in a clause by itself, to say that this output must be true for a cnf to accept.

This reduction preserves satisfiability, because circuits which have two different satisfiability-assignments will be satisfiable after the reduction. This reduction also preserves non-satisfiability, as circuits which have less then two satisfying assignments cannot get both instances of the reduction to be true, while preventing that corresponding input variables are equal.

The time-complexity of the reduction is the same as for `CSAT_to_SAT`. Note that the nested while loop goes from some gate-number to the last agte-numer, and catches up by one for each iteration, thus produces a time-complexity which is linear in the number of gates, and thus squred in the number of inputs, following the same logic as put forward in the end of exercise 1 above.

Exercise 3: Test the implementations, also with your own test-cases

I have supplied a test-class in the file `test_circuitsat.py`. I have constructed eleven additional tests, each used in `test_circuitsat.py` on both the implementation for exercise one and two. The files for the custom tests are located in the folder `testfiles`.